Programación Funcional en Haskell

Primera parte

Paradigmas de Lenguajes de Programación

Departamento de Ciencias de la Computación
Universidad de Buenos Aires

17 de agosto de 2023

Repaso: usando GHCi

Cómo empezar:

```
$
```

Repaso: usando GHCi

Cómo empezar:

```
$ ghci
Loading ...
Prelude>
```

Repaso: usando GHCi

Cómo empezar:

```
$ ghci
Loading ...
Prelude>:q
Leaving GHCi.
$
```

Repaso: usando GHCi

Cómo empezar:

```
$ ghci
Loading ...
Prelude>:q
Leaving GHCi.
$ ghci test.hs
Loading ...
[1 of 1] Compiling Main ( test.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Repaso: usando GHCi

Cómo empezar:

```
$ ghci
Loading ...
Prelude>:q
Leaving GHCi.
$ ghci test.hs
Loading ...
[1 of 1] Compiling Main ( test.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Otros comandos útiles:

- Para recargar: `:r`
- Para cargar otro archivo: `:l archivo.hs`
- Para conocer el tipo de una expresión: `:t True`

Aplicación parcial

Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int
prod x y = x * y
```

Aplicación parcial

Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int
prod x y = x * y
```

Definimos `doble x = prod 2 x`

1. ¿Cuál es el tipo de `doble`?

Aplicación parcial

Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int
prod x y = x * y
```

Definimos `doble x = prod 2 x`

1. ¿Cuál es el tipo de `doble`?
2. ¿Qué pasa si cambiamos la definición por `doble = prod 2`?

Aplicación parcial

Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int
prod x y = x * y
```

Definimos `doble x = prod 2 x`

1. ¿Cuál es el tipo de `doble`?
2. ¿Qué pasa si cambiamos la definición por `doble = prod 2`?
3. ¿Qué significa `(+) 1`?

Aplicación parcial

Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int
prod x y = x * y
```

Definimos `doble x = prod 2 x`

1. ¿Cuál es el tipo de `doble`?
2. ¿Qué pasa si cambiamos la definición por `doble = prod 2`?
3. ¿Qué significa `(+) 1`?
4. Definir las siguientes funciones de forma similar a `(+)1`:

Aplicación parcial

Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int
prod x y = x * y
```

Definimos `doble x = prod 2 x`

1. ¿Cuál es el tipo de `doble`?
2. ¿Qué pasa si cambiamos la definición por `doble = prod 2`?
3. ¿Qué significa `(+) 1`?
4. Definir las siguientes funciones de forma similar a `(+)1`:
   - `triple :: Float -> Float`

Aplicación parcial

Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int
prod x y = x * y
```

Definimos `doble x = prod 2 x`

1. ¿Cuál es el tipo de `doble`?
2. ¿Qué pasa si cambiamos la definición por `doble = prod 2`?
3. ¿Qué significa `(+) 1`?
4. Definir las siguientes funciones de forma similar a `(+)1`:
   - `triple :: Float -> Float`
   - `esMayorDeEdad :: Int -> Bool`

Funciones muy útiles

Ejercicios

1. Implementar y dar los tipos de las siguientes funciones:

Funciones muy útiles

Ejercicios

1. Implementar y dar los tipos de las siguientes funciones:
   a. `(.)` que compone dos funciones. Por ejemplo:
      `((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.

Funciones muy útiles

Ejercicios

1. Implementar y dar los tipos de las siguientes funciones:
   a. `(.)` que compone dos funciones. Por ejemplo:
      `((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
   b. `flip` que invierte los argumentos de una función. Por ejemplo:
      `flip (\x y -> x - y) 1 5` devuelve 4.

Funciones muy útiles

Ejercicios

1. Implementar y dar los tipos de las siguientes funciones:
   a. `(.)` que compone dos funciones. Por ejemplo:
      `((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
   b. `flip` que invierte los argumentos de una función. Por ejemplo:
      `flip (\x y -> x - y) 1 5` devuelve 4.
   c. `($)` que aplica una función a un argumento. Por ejemplo:
      `id $ 6` devuelve 6.

Funciones muy útiles

Ejercicios

1. Implementar y dar los tipos de las siguientes funciones:
   a. `(.)` que compone dos funciones. Por ejemplo:
      `((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
   b. `flip` que invierte los argumentos de una función. Por ejemplo:
      `flip (\x y -> x - y) 1 5` devuelve 4.
   c. `($)` que aplica una función a un argumento. Por ejemplo:
      `id $ 6` devuelve 6.
2. ¿Qué hace `flip ($) 0`?

Funciones muy útiles

Ejercicios

1. Implementar y dar los tipos de las siguientes funciones:
   a. `(.)` que compone dos funciones. Por ejemplo:
      `((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
   b. `flip` que invierte los argumentos de una función. Por ejemplo:
      `flip (\x y -> x - y) 1 5` devuelve 4.
   c. `($)` que aplica una función a un argumento. Por ejemplo:
      `id $ 6` devuelve 6.
2. ¿Qué hace `flip ($) 0`?
3. ¿Y `(==0) . (flip mod 2)`?

Ejercicios

Definir las siguientes funciones. Precondición: la lista tiene al menos un elemento.

1. `maximo :: Ord a => [a] -> a`
2. `minimo :: Ord a => [a] -> a`
3. `listaMasCorta :: [[a]] -> [a]`

Ejercicios

Las definiciones anteriores son esencialmente iguales.
Veamos cómo generalizar la idea.

1. Definir `mejorSegun :: (a -> a -> Bool) -> [a] -> a`, teniendo también como precondición que la lista tiene al menos un elemento, de manera que `maximo = mejorSegun (>)`.
2. Reescribir `minimo` y `listaMasCorta` usando `mejorSegun`.

Listas

Hay varias formas de definir una lista:

- **Por extensión**
  Esto es, dar la lista explícita, escribiendo todos sus elementos.
  Por ejemplo: `[4, 3, 3, 4, 6, 5, 4, 5, 4, 5]`.

Listas

Hay varias formas de definir una lista:

- **Por extensión**
  Esto es, dar la lista explícita, escribiendo todos sus elementos.
  Por ejemplo: `[4, 3, 3, 4, 6, 5, 4, 5, 4, 5]`.

- **Secuencias**
  Son progresiones aritméticas en un rango particular.
  Por ejemplo: `[3..7]` es la lista que tiene todos los números enteros entre 3 y 7, mientras que `[2, 5..18]` es la lista que contiene 2, 5, 8, 11, 14 y 17.

Listas

Hay varias formas de definir una lista:

- **Por extensión**
  Esto es, dar la lista explícita, escribiendo todos sus elementos.
  Por ejemplo: `[4, 3, 3, 4, 6, 5, 4, 5, 4, 5]`.

- **Secuencias**
  Son progresiones aritméticas en un rango particular.
  Por ejemplo: `[3..7]` es la lista que tiene todos los números enteros entre 3 y 7, mientras que `[2, 5..18]` es la lista que contiene 2, 5, 8, 11, 14 y 17.

- **Por comprensión**
  Se definen de la siguiente manera:
  ```
  [expresión | selectores, condiciones]
  ```
  Por ejemplo: `[(x,y) | x <- [0..5], y <- [0..3], x+y==4]` es la lista que tiene los pares (1,3), (2,2), (3,1) y (4,0).

Listas infinitas

Haskell también nos permite trabajar con **listas infinitas**.

Listas infinitas

Haskell también nos permite trabajar con **listas infinitas**.

Algunos ejemplos:

- `naturales = [1..]`

  1, 2, 3, 4, ...

Listas infinitas

Haskell también nos permite trabajar con **listas infinitas**.

Algunos ejemplos:

- `naturales = [1..]`

  1, 2, 3, 4, ...

- `multiplosDe3 = [0,3..]`

  0, 3, 6, 9, ...

Listas infinitas

Haskell también nos permite trabajar con **listas infinitas**.

Algunos ejemplos:

- `naturales = [1..]`
  1, 2, 3, 4, ...
- `multiplosDe3 = [0,3..]`
  0, 3, 6, 9, ...
- `repeat ‘‘hola’’`
  “hola”, “hola”, “hola”, “hola”, ...

Listas infinitas

Haskell también nos permite trabajar con **listas infinitas**.

Algunos ejemplos:

- `naturales = [1..]`
  1, 2, 3, 4, ...
- `multiplosDe3 = [0,3..]`
  0, 3, 6, 9, ...
- `repeat ‘‘hola’’`
  “hola”, “hola”, “hola”, “hola”, ...
- `primos = [n | n <- [2..], esPrimo n]`
  (asumiendo `esPrimo` definida) 2, 3, 5, 7, ...

Listas infinitas

Haskell también nos permite trabajar con **listas infinitas**.

Algunos ejemplos:

- `naturales = [1..]`
  1, 2, 3, 4, ...
- `multiplosDe3 = [0,3..]`
  0, 3, 6, 9, ...
- `repeat ‘‘hola’’`
  “hola”, “hola”, “hola”, “hola”, ...
- `primos = [n | n <- [2..], esPrimo n]`
  (asumiendo `esPrimo` definida) 2, 3, 5, 7, ...
- `infinitosUnos = 1 : infinitosUnos`
  1, 1, 1, 1, ...

Listas infinitas

Haskell también nos permite trabajar con **listas infinitas**.

Algunos ejemplos:

- `naturales = [1..]`
  1, 2, 3, 4, ...
- `multiplosDe3 = [0,3..]`
  0, 3, 6, 9, ...
- `repeat ‘‘hola’’`
  “hola”, “hola”, “hola”, “hola”, ...
- `primos = [n | n <- [2..], esPrimo n]`
  (asumiendo `esPrimo` definida) 2, 3, 5, 7, ...
- `infinitosUnos = 1 : infinitosUnos`
  1, 1, 1, 1, ...

¿Cómo es posible trabajar con listas infinitas sin que se cuelgue?


HASKELL
"I can't believe I eval'ed the whole thing."
ACTIVITIES: NONE
SPORTS: NONE
HONORS: NONE

Evaluación lazy

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs

infinitosUnos :: [Int]
infinitosUnos = 1 : infinitosUnos

nUnos :: Int -> [Int]
nUnos n = take n infinitosUnos
```

Evaluación lazy

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs

infinitosUnos :: [Int]
infinitosUnos = 1 : infinitosUnos

nUnos :: Int -> [Int]
nUnos n = take n infinitosUnos
```

- Mostrar los pasos necesarios para reducir `nUnos 2`.

  `nUnos 2` $\rightarrow$ `take 2 infinitosUnos` $\rightarrow$ `take 2 (1:infinitosUnos)` $\rightarrow$ `1 : take (2-1) infinitosUnos` $\rightarrow$ ...

Evaluación lazy

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs

infinitosUnos :: [Int]
infinitosUnos = 1 : infinitosUnos

nUnos :: Int -> [Int]
nUnos n = take n infinitosUnos
```

- Mostrar los pasos necesarios para reducir `nUnos 2`.

  `nUnos 2` $\rightarrow$ `take 2 infinitosUnos` $\rightarrow$ `take 2 (1:infinitosUnos)` $\rightarrow$ `1 : take (2-1) infinitosUnos` $\rightarrow$ ...
- ¿Qué sucedería si usáramos otra estrategia de reducción?

Evaluación lazy

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs

infinitosUnos :: [Int]
infinitosUnos = 1 : infinitosUnos

nUnos :: Int -> [Int]
nUnos n = take n infinitosUnos
```

- Mostrar los pasos necesarios para reducir `nUnos 2`.

  `nUnos 2` $\rightarrow$ `take 2 infinitosUnos` $\rightarrow$ `take 2 (1:infinitosUnos)` $\rightarrow$ `1 : take (2-1) infinitosUnos` $\rightarrow$ ...
- ¿Qué sucedería si usáramos otra estrategia de reducción?
- Si para algún término existe una reducción finita, entonces la estrategia de reducción lazy termina.

Esquemas de recursión sobre listas: filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) =
      if p x
      then x : filter p xs
      else filter p xs
```

Esquemas de recursión sobre listas: filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) =
      if p x
      then x : filter p xs
      else filter p xs
```

Ejercicios

Definir usando `filter`

Esquemas de recursión sobre listas: filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) =
      if p x
      then x : filter p xs
      else filter p xs
```

Ejercicios

Definir usando `filter`

1. `deLongitudN :: Int -> [[a]] -> [[a]]`

Esquemas de recursión sobre listas: filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) =
      if p x
      then x : filter p xs
      else filter p xs
```

Ejercicios

Definir usando `filter`

1. `deLongitudN :: Int -> [[a]] -> [[a]]`
2. `soloPuntosFijosEnN :: Int -> [Int->Int] -> [Int->Int]`
   Dados un número n y una lista de funciones, deja las funciones que al aplicarlas a n dan n.

Esquemas de recursión sobre listas: map

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Esquemas de recursión sobre listas: map

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Ejercicio

Definir usando `map`:

Esquemas de recursión sobre listas: map

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Ejercicio

Definir usando `map`:

1. `shuffle :: [Int] -> [a] -> [a]` que, dada una lista de índices $[i_1, \ldots, i_n]$ y una lista ℓ, devuelve la lista $[\ell_{i_1}, \ldots, \ell_{i_n}]$.

Esquemas de recursión sobre listas: map

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Ejercicio

Definir usando `map`:

1. `shuffle :: [Int] -> [a] -> [a]` que, dada una lista de índices $[i_1, \ldots, i_n]$ y una lista ℓ, devuelve la lista $[\ell_{i_1}, \ldots, \ell_{i_n}]$.
2. `reverseAnidado :: [[Char]] -> [[Char]]` que, dada una lista de strings, devuelve una lista con cada string dado vuelta y la lista completa dada vuelta. Por ejemplo: `reverseAnidado [‘‘quedate’’, ‘‘en’’, ‘‘casa’’]` devuelve `[‘‘asac", ‘‘ne’’, ‘‘etadeuq’’]`.

Esquemas de recursión sobre listas: map

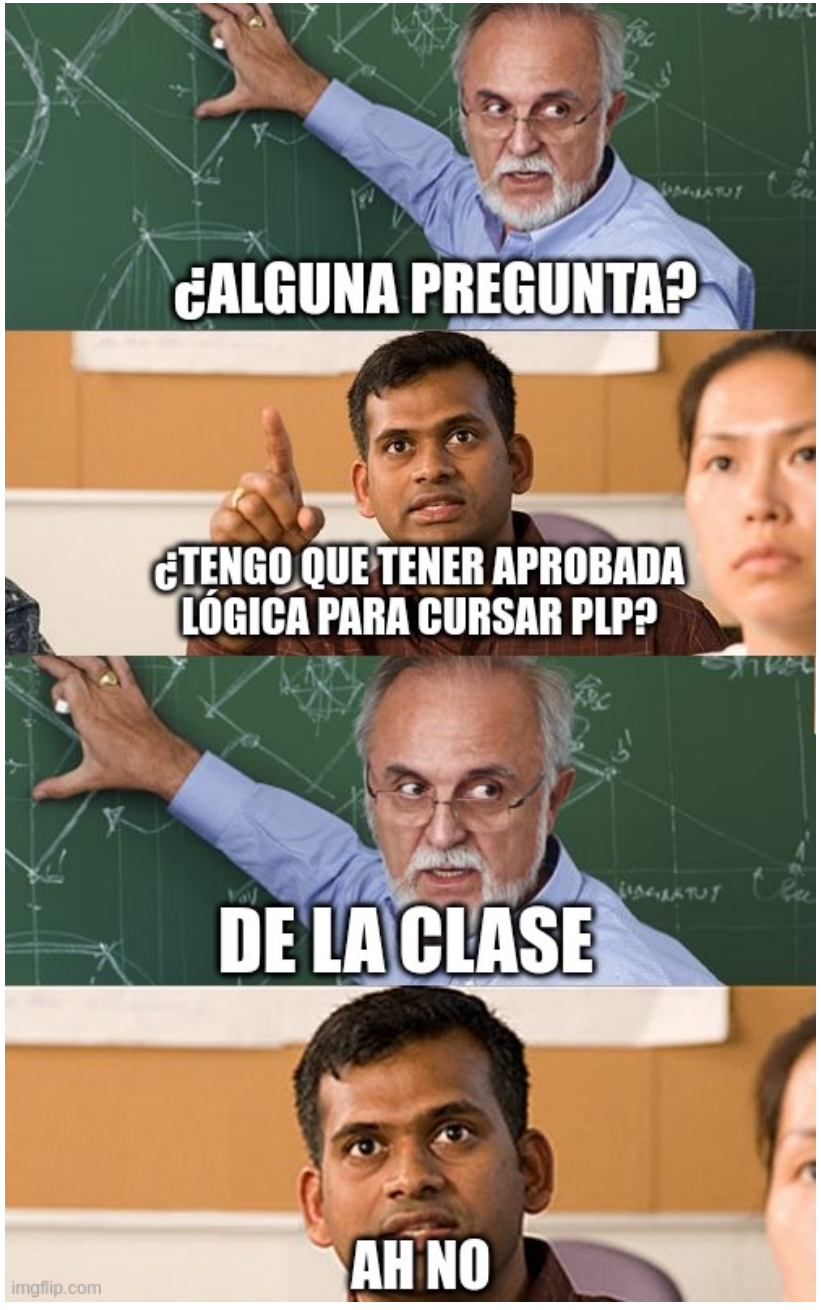
```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Ejercicio

Definir usando `map`:

1. `shuffle :: [Int] -> [a] -> [a]` que, dada una lista de índices $[i_1, \ldots, i_n]$ y una lista ℓ, devuelve la lista $[\ell_{i_1}, \ldots, \ell_{i_n}]$.
2. `reverseAnidado :: [[Char]] -> [[Char]]` que, dada una lista de strings, devuelve una lista con cada string dado vuelta y la lista completa dada vuelta. Por ejemplo: `reverseAnidado [‘‘quedate’’, ‘‘en’’, ‘‘casa’’]` devuelve `[‘‘asac", ‘‘ne’’, ‘‘etadeuq’’]`.
3. `paresCuadrados :: [Int] -> [Int]` que, dada una lista de enteros, devuelve una lista con los cuadrados de los números pares.

¿Preguntas?


¿ALGUNA PREGUNTA?
¿TENGO QUE TENER APROBADA LÓGICA PARA CURSAR PLP?
DE LA CLASE
AH NO
imgflip.com